**Notas git y github** (/OneDrive/Documents/Universidad/Cursos/Git y Github/Proyecto1)

Git es creado por a la fundación que hace Linux. es el que guarda las versiones, es el más famoso en todo el mundo

Git init 🡪 arranca el repositorio show

Git add 🡪 arranca el archivo

Git commit -m “version1” 🡪 envía los últimos cambios a la base de versiones

Git status 🡪 ver es estatus de la base de datos

Git show 🡪 muestra todos los cambios que se hallan hecho

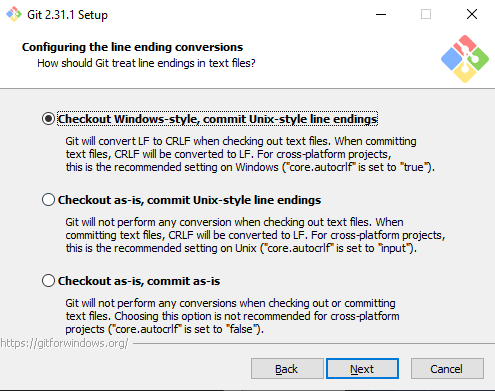
Git log (nombre de archivo) 🡪 muestra todo

Git push 🡪 envía a otro repositorio remoto

Gti pull 🡪 trae de un repositorio remoto

SSL: librerías que cifra para seguridad

Windows y Linux graban el enter de manera distinta. 1. lo hace git, 2. A lo de dios, 3. Yo 😊



**Editores de código y archivos binarios:**

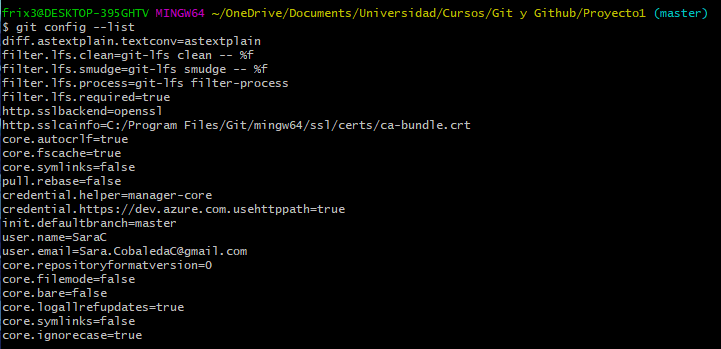
Es distinto un archivo de texto plano a un archivo binario que parece de texto. Por ejemplo, un Word no es un archivo de texto plano, sino un archivo binario, pero un txt si es un archivo de texto plano, que es tal cual como se ve. Se va a utilizar Visual Studio Code.

La estructura de archivos es distinta en Windows, Linux y mac. En Windows la mayúscula no importa y puedes colocar todo pegado, pero en Linux y Mac si importa.

**Inicio:**

Git init para crear una carpeta dentro de proyecto uno que será donde se guarden los cambios atómicos del proyecto. Se crea un txt en la carpeta de proyecto, y con git status se puede ver el status del proyecto. Después se agrega el nuevo archivo con git add historia.txt, pero todavía no ha sido enviado al repositorio, con git rm –catch historia.txt se puede sacar el archivo. El “catch” significa que está en memoria RAM, que aún no está en la base de datos. Con git commit -m “ese es la primera versión” se envía al repositorio con un mensaje, pero hay que decirle quien soy, para saber quién realizo el cambio.

Con git congif va a saltar todas las configuraciones de git, y con git config –list se ve la configuración por defecto. Con git config –list –show—origin se ve donde se guardan las configuraciones. Con git config –global user.name “Sara” se configura el usuario de git. Con git config –global user.email “[Sara.CobaledaC@gmail.com](mailto:Sara.CobaledaC@gmail.com)”



Ahora si se pueden ver las configuraciones. Se vuelve a ejecutar el git commit -m “ese es la primera versión” y ahora si se va a agregar al repositorio.

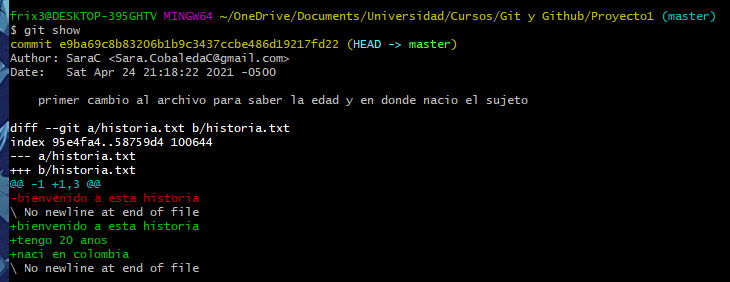
Con cat historia.txt, se ve el contenido del archivo, y con code historioa.txt se puede abrir con visual studio code y editarlo. Con add . Se agregan todos los archivos de la carpeta.

Con git log historia.txt me muestra todos los cambios que sucedieron en el archivo. El numero largo es el tag, que es la indicación de donde estoy en este momento, el nombre de ese commit, y cuando dice HEAD -> master significa que es la versión más reciente, muestra el autor, la fecha, y los cambios hechos antes.

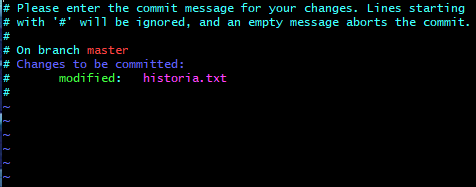
(con un solo guion se usan las letras como -al, el doble guion significa una palabra)

**Analizar cambios:**

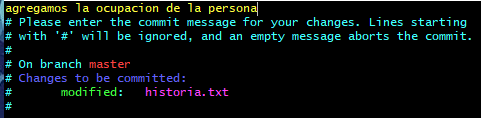
Con git show historia.txt muestra los cambios realizados en el archivo. Muestra la versión más reciente, el mensaje del commit, y muestra una comparación de la versión vieja y una versión nueva. Los cambios aparecen en verde y rojo, y lo que no cambio aparece en blanco.



Es una buena práctica colocar mensajes. Con Esc+Shift+z+z se sale del menú de donde se está agregando el mensaje si no se agregó en un principio.



El menú que aparece cuando no se le agrega mensaje es un editor de código basado en línea de comandos, se llama vin. # significa que es un comentario, con Esc+i va a permitir agregar texto. Nunca se envía un commit vacío.



Con git log historia.txt, se ve que ya tiene varios commits. Con git diff y dos números de dos commits diferentes, compara lo que hay entre esas dos versiones. La versión que se coloque de primeras la va a tomar como la original.

**Ciclo básico de trabajo de git (staging y los repositorios):**

Cuando se le da git add, el archivo pasa a estar en staging (está en memoria RAM, un área de preparación) y está esperando a ser enviado al repositorio, o ser removido. Para enviarlo al repositorio se usa git commit (cada commit es una nueva versión del archivo), el repositorio se llama master, donde van a estar todos los cambios que se realicen.

Untracked: sin rastrear

Tracked: el archivo está siendo rastreado y entro al staging con git add, o ya se agregó al repositorio.

Para traer los últimos cambios se utiliza check out, o los cambios que uno quiera hacia la carpeta de archivos locales.

Las ramas es romper en pedacitos el código para que cada uno trabaje en un área.

**Que es una brach (rama) y como funciona un merge en git**:

Por defecto se encuentra en la rama master, en donde se almacenan los cambios de los archivos con cada commit, pero para hacer pruebas se crean ramas. Básicamente se copia la versión en donde se va a trabajar ya sea actual o anterior de la rama master, y se le pone un nombre, y se continua con los commits, esa rama será diferente en código y contenido a la rama master (rama development).

Si se quiere arreglar un bug de la versión actual, se crea una rama llamada hotfix o bugfixing en donde se realizan pruebas para tratar de solucionar el bug, un merge se conoce como la unión de los cambios de una rama con otra.

Al final de los experimentos se pueden unir con la rama master.

Hay que tener cuidado cuando se realiza un merge para evitar conflictos/

Con git reset se puede volver a una versión anterior, existen dos, el duro y es suave, con git reset hard se le da reste a todo y es el que es más común en su uso, pero con git reset soft se vuelve a la versión anterior, pero lo que está en staging sigue en staging, pero cambia el directorio de trabajo.



Hay que tener cuidado con hard, porque realmente borra todo lo que se realizó antes. Una forma de volver al pasado. Si se crea una carpeta, a git solo le importan los archivos así que la guarda como una ruta.

Con git log --stat se pueden ver los cambios específicos que se realizaron en que archivos a partir del commit y con la letra q se puede salir de ahí. Con git checkout y el numero de la versión y el nombre del archivo se puede ver cómo era el archivo en esa versión en específico, al poner el comando git status se ve que el archivo está en memoria RAM, pero si se hace un commit se pierde los cambios de antes porque envía ese archivo al repositorio.



Con git checkout master historia.txt se regresa a la última versión. Esta es la forma de volver a una versión anterior.

Básicamente se volvió a una versión anterior y se empezó de nuevo desde ahí.

Git rm:

ayuda a eliminar archivos de Git sin eliminar su historial del sistema de versiones. Esto quiere decir que si necesitamos recuperar el archivo solo debemos “viajar en el tiempo” y recuperar el último commit antes de borrar el archivo en cuestión.

git rm --cached: Elimina los archivos del área de Staging y del próximo commit, pero los mantiene en nuestro disco duro.

git rm --force: Elimina los archivos de Git y del disco duro. Git siempre guarda todo, por lo que podemos acceder al registro de la existencia de los archivos, de modo que podremos recuperarlos si es necesario (pero debemos usar comandos más avanzados).

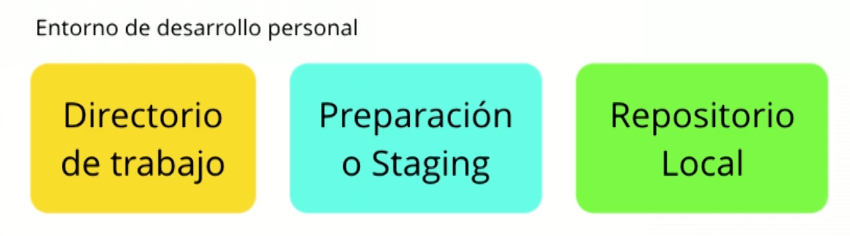
Git reset:

git reset --soft: Borramos todo el historial y los registros de Git, pero guardamos los cambios que tengamos en Staging, así podemos aplicar las últimas actualizaciones a un nuevo commit.

git reset --hard: Borra todo, todo todito, absolutamente todo. Toda la información de los commits y del área de staging se borra del historial.

git reset HEAD: Este es el comando para sacar archivos del área de Staging. No para borrarlos ni nada de eso, solo para que los últimos cambios de estos archivos no se envíen al último commit, a menos que cambiemos de opinión y los incluyamos de nuevo en staging con git add, por supuesto.

**Flujo de trabajo con git:**



Staging es el lugar donde se enviarán los archivos antes de mandarlos al repositorio local, el repositorio local es la base de datos con toda la historia del proyecto. Con git add los manda al staging, para enviarlo al repositorio se usa git commit y hay que agregarle un mensaje.

Para trabajar en grupo, hay que tener un repositorio remoto, hay traer datos de un repositorio remoto se hace con git clone url, que clona todo el historial en el repositorio local y atrae todos los archivos al directorio de trabajo. Cuando ya todo está listo se envía al repositorio remoto se utiliza con git push.



Para traer información actualizada se utiliza git fetch, y lo trae al repositorio local, y hay que fusionarlo con lo que se clono y está en el directorio de trabajo, para eso se utiliza git merge. Pero para que se hagan ambas cosas se utiliza git pull.

**Introducción a las ramas:**

Las ramas son formas de hacer cambios sin afectar la rama maestra que es donde esta toda la historia de commits, el commit más reciente es la cabecera.

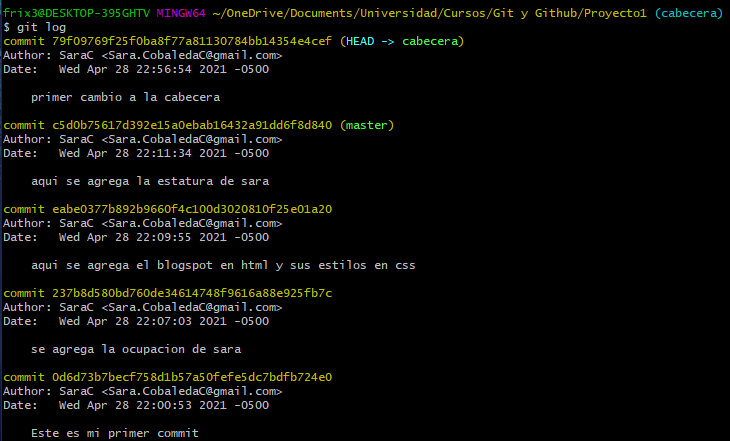
Supongamos que vamos a crear una rama llamada cabecera, esto se hace haciendo una copia del ultimo commit en otro lado, y todos los cambios de esta rama no los va a ver la cabecera y realizando cambios sobre ella, y no va a volver a la rama master hasta que se hallan fusionado ambas ramas.

(git checkout head, se devuelve a head del master :v)

Se le hacen cambios al blogspot y a sus estilos, con git commit -am automáticamente hace el git add de los cambios, solo funciona con archivos que ya se les haya dado add previamente, y se envían los cambios al repositorio con su respectivo mensaje.

La rama se va a crear desde el lugar donde estoy, para crear la rama se escribe git branch cabecera, si se le da git show va a mostrar un head que apunta al master y a la cabecera, o sea que está pegado a dos ramas distintas, pero sigo en la rama master. Con git checkout cabecera se mueve a la rama llamada cabecera, y con git status va a decir que estamos en cabecera.

A continuación hacemos cambios al HTML, y le damos git status para ver si detecto el cambio y se le da git add blogpost.html y git commit -m ”estructura inicial de la cabecera”. Con git show va a mostrar que estamos en head que apunta cabecera, y con git log me sigue mostrando todos los commits, y dice cual es de que rama.



Si me muevo de nuevo a la rama master con git checkout master el archivo HTML va a perder los cambios que hayamos hecho en el mientras estábamos en la rama.

Head es un indicador de cual versión de commit estoy viendo.

**Fusión de ramas con git merge:**

En primer lugar, vamos a hacer un nuevo commit en la rama cabecera cambiando el título, hay que hacer cambios a ambos archivos, al CSS y al HTML, y otro en la rama master cambiando le contenido. Cuando se hace un merge, la rama adyacente desaparece y la rama master continua, pero no aparece una nueva rama.

Con git commit -am “finalizada la cabecera con diseño azul”, subimos los cambios a la rama cabecera, después hacemos un párrafo adicional al HTML en la rama master y se los agrego con git commit -am “agregado el contenido adicional del blog” a la rama master. Aquí se puede ver como los archivos cambian según a donde se haga el git checkout.

Un merge se hace desde el lugar en donde estoy y es a donde me va a pegar, por ejemplo, si estoy en cabecera y hago el merge desde ahí, la rama cabecera se va a convertir en la nueva rama master. Para hacer el merge hay que poner el head en master, y desde ahí hacer un merge de cabecera, lo que va a crear un commit con ambas versiones y las va a funcionar.

Con git Branch se puede saber el nombre de las ramas, con Git checkout master se devuelve de nuevo a la rama master y ahí se coloca el comando git merge cabecera, y va a abrir vin para colocarle un mensaje. Aquí va a aparecer un mensaje que dice que esta fusionando ambas versiones, y con git log se pueden ver los commit que se realizaron en la rama cabecera. al ver los archivos HTML y CSS, se observan los cambios realizados en master y en la rama cabecera.

**¿Qué hacer cuando hay algún conflicto?**

Cuando se realizó el merge de la rama cabecera a la rama master, la rama cabecera realmente no se borró, porque git no borra nada, y si se recarga el último commit efectivamente aparece la cabecera. Algo que se puede hacer es hacer el merge al revés, ósea traer lo de master a cabecera, o sea que el ultimo commit de master se pegó a cabecera. Ahora se va a simular un conflicto modificando las mismas líneas en master y cabecera, por ejemplo, cambiando el título y el color, adicionalmente en master se va a agregar un párrafo adicional, y se hace el commit en cada rama.

El párrafo no debería generar problemas, pero las otras cosas sí. Se le da git merge cabecera desde master, ahí van a aparecer los conflictos encontrados, por lo que no se tiene un merge completo (para que se complete un merge tiene que ocurrir un commit)

La primera forma de resolver el conflicto es borrar los mensajes que pone git en el código, y escoger alguna de las dos versiones, en code aparece una interfaz en donde se puede escoger automáticamente cuales de los cambios se van a quedas, esta interfaz es de git (por favor guardar con Ctrl+s). Final mente se le da un git commit -am ”solución de conflicto de las ramas”.

Una vez los cambios están hechos se hace un git checkout cabecera, y aparecerá la versión master “rebelde”. Git va a atraer ambos códigos y los va a comparar para escoger, pero siempre va a quedar registro de lo que se hizo, y quien lo hizo.

**Uso de github:**

GitHub es una página web que tiene un servidor de git gigante en línea, se le dice la red social de los programadores. Aquí se creará un nuevo repositorio llamado hyperblog y se le coloca alguna descripción, después preguntara por readme que es un archivo que se va a ver en el instante que entre al repositorio en github que explica el repositorio, es una muy buena práctica.

Existen muchas licencias con las que se puede crear el código.

Raw muestra el código plano del archivo, blame muestra los cambios y quien los hizo, y History muestra la historia del archivo, como in git log. Para poner un repositorio en la memoria del pc de git a github, se selecciona la opción de clone or download y se copia el enlace, después se vuelve a la consola de git y se le dice que agregue un origen remoto con el comando git remote add origin (enlace copiado de github).

Con el comando git remote muestra algo llamado origin, con git remote -v muestra cuales son los enlaces para el fetch y para push. Con git push origin master se envía la rama master al origen, en Windows abrirá una nueva ventana donde se logea en GitHub. Aquí aparece un error que básicamente dice que hay que traer el repositorio remoto y fusionarlo con master.

Para traer es git pull origin master, y va a aparecer una advertencia que no hay commits comunes, como si fueran historias distintas, con git pull origin master --allow-unrelated-histories se fuerza la fusión de las historias, y aparecerán los archivos en la memoria local, pero en github todo sigue igual. Ahora si se le pone git push origin master, y sube todos los archivos a github con todo y los commits.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Se pueden editar los archivos desde github, en donde ya hay una interfaz gráfica bien linda donde pregunta la descripción y en que rama se almacena, pero esos cambios no se verán desde la consola de git, para eso es git pull origin master. Con git log muestra todo y origin es básicamente lo que está en github.

**Llaves públicas y privadas:**

Llaves públicas y privadas es un algoritmo también conocido como cifrado asimétrico de un solo camino, que sirve para enviar las llaves para desencriptar los mensajes privados. Funciona a través de una llave pública y privada que están vinculadas matemáticamente de tal manera que lo que se cifre con la llave publica, solo lo va a ver la llave privada.

Entonces lo que se hace es enviar la llave publica por internet, y esa llave publica va a permitir cifrar el mensaje. La llave publica puede ser conocida por todos, pero lo importante es que sigue conectada a la llave privada y nadie debería conocer la llave privada. Entonces se copia la llave publica y se cifra el mensaje, para generar un nuevo mensaje que no importa si es interceptado junto con la llave publica, porque no puede ser descifrado, para desencriptar el mensaje, se utiliza la llave privada. Así funcionan todas las finanzas del mundo.

**Configuración de llaves SSH en local:**

Cuando uno se logea en un lugar, el usuario y la contraseña se guardan en el entorno local, si se roban el pc, se está en riesgo de password cracking, para evitar eso hay que crear un entorno de llaves públicas y privadas.

En el entorno local hay que crear una llave privada y una publica, y la llave publica se envía a github, para que en determinado repositorio utilice esa llave publica, ahora se conectara por un protocolo SSH en vez de https, ese protocolo se utiliza para conectarse romanamente a servidores. En la primera conexión github va a enviar una llave publica cifrada con la llave publica que se acaba de enviar. o sea que hay un doble camino cifrado de conexión con git hub.

Diagrama

Descripción generada automáticamente con confianza baja

Las llaves SSH no son por repositorio o por proyecto, sino por persona. Para eso hay que estar en la carpeta home y ahí se creara la llave. Ahora se cambiará la configuración de git, porque el correo de github es diferente al correo registrado en git, se coloca el comando git config -l para ver la configuración actual y para cambiar el email se usa git config --global user.email “correo en github”.

Para crear la llave se escribe ssh-keygen -t rsa -b 4096 -C “correo de github”, -t especifica el algoritmo que se va a utilizar y el rsa es un algoritmo para crear llaves, existen muchos, el -b que especifica la complejidad de la llave y 4096 es la complejidad, con -C se especifica el correo al cual va a estar conectada la llave. Ahora va a preguntar en donde se va a guardar la llave, simplemente se le da enter. A continuación, pide la contraseña adicional de texto a las llaves, una passphrase.

Ahora muestra en donde se guardó la llave pública y la llave privada, el fingerprint y el randomart que es para saber que la llave es de verdad. Las llaves se pueden ver con visual estudio code, por lo que ahora se copia y se pega la llave publica en github.

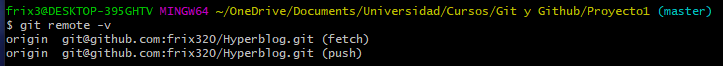
En window o Linux hay que revisar que el servidor ssh este prendido, es para que el sistema operativo revise que las llaves estén funcionando. Para eso se pone eval $(ssh-agent -s) y va a salir agen(el servidor ssh está corriendo) pid(process id) y un número, que demuestra que está funcionando bien. Ahora hay que decirle al servidor que la llave existe, para eso hay que poner ssh-add ~/.ssh/id\_rsa (~ es una variable que tiene el nombre de la carpeta home)

No se le agrega la que termina en .pub, que es la pública, sino que se agrega la llave privada.

**Conexión a github con SSH:**

Cada computadora debe tener su propia llave privada, o sea que hay que crear una llave privada por cada repositorio. Vamos a copiar la llave publica, y en github en settings en el apartado de SSH and GPG keys y se crea una nueva llave SSH, se le da un nombre a la llave y se pega la llave pública, finalmente pide la contraseña y se le da continuar.

Después se entra el repositorio en github, y se le da click en clone or download, se le da el botón use SSH y se copia el enlace. En el repositorio local, se le da git remote -v para ver la url de los repositorios, y hay que cambiar la url de origin, para eso se le da a git remote set-url origin (se pega la url SSH). Para verificar si funciona hay que hacer cambios en los archivos y enviarlos.



Antes de hacer un commit, hay que traer la última versión del servidor con git pull, ahora está preguntando la autenticidad de github, simplemente se le da yes. Ahora se le da git pull origin master, para traer el repositorio remoto y fusionarlo con la rama actual que es master, eso trae los cambios, y con git status va a mostrar que se realizaron cambios (porque se le agrago un 2.0 al html localmente), ahora se hace un git commit -am “una versión del hiperblog”, por si acaso se le va a volver a dar git pull origin master, y finalmente se envía el cambio son git push origin master.

En github debería mostrar el cambio realizado en el archivo html.

**Tags y versiones en git y github:**

Con git log --all muestra TODOS los commits, y con git log --all --graph que muestra como funcionan las ramas, git log --all --graph --decorate --oneline va a mostrar todo mucho más comprimido. Existe un truco para colocar un alias a los comandos que es: alias (nombre del alias)=”git log --all --graph --decorate --oneline”. Los alioas quedan guardados en el entorno local.

Un tag es como la división de versiones que existen en los programas, algo así como 1.0 o 1.1 y esas cosas. Para crear un tag se copia el hush (el numerito amarillo con el que se identifican los commits hay uno largo y uno corto, se puede pegar el corto si no hay ninguno igual) y se pone el comando git tag -a v0.1 -m “Resultado de las primeras clases del curso” (se pega el hush). Con git tag va a mostrar todos los tags, pero con git show-ref --tags va a mostrar que tag este asignado a que commit, pero si se le da history, el número que va a mostrar es diferente, porque el que muestra es la referencia al momento en el que se crea el tag.

Los tags son muy útiles en github, y hay que enviarlos al repositorio remoto. Para eso se pone git pull origin master para traer el repositorio remoto, y para enviar los tags se pone git push origin –tags.

Si en algún momento se crea mal un tag, se puede eliminar desde la interfaz de github, pero desde la consola de git hay que poner git tag -d (nombre del tag como v0.1), si se envían de nuevo los tags al repositorio remoto en github no se van a borrar automáticamente, porque funcionan como releases. Para poder borrarlo bien se le da en la consola git push origin :refs/tags/(nombre del tag como v0.1) .

**Manejo de ramas en github:**

Cuando se envió por primero vez la rama master al repositorio remoto en github, se pasó por la faja la rama cabecera, por lo que en github solo está la rama master. Para agregar las ramas como cabecera a github se le da el comando git pull origin master, después se mueve a la rama cabecera con git checkout cabecera y se da git push origin cabecera.

Con git show-branch muestra las ramas y su historia, pero con git show-branch --all va a mostrar mas detalles, y con gitk abre un software en donde se encuentra visualmente las ramas.

A continuación, se crean dos ramas adicionales, para eso se vuelve de nuevo a master para que las ramas salgan de ahi y se crea git branch header y git branch footer. Después se envían al repositorio remoto con git push origin header y git push origin footer.

**Configurar múltiples colaboradores en un repositorio de GitHub:**

Supongamos que va a haber un programador que haga el encabezado en la rama header y otro que va a trabajar en el cuerpo en la rama footer, para empezar, hay que configurar el entorno de trabajo de cada programador (correos, nombre y esas cosas) y en vez de hacer in git init, tiene que traerse el entorno hyperblog de github.

Para unirse a un proyecto en vez de empezar de 0 hay que poner git clone (url de github ya sea de https o SSH), si el repositorio es público, no va a pedir usuario ni contraseña. El hecho de clonar el repositorio no quiere decir que el se pueda enviar cambios al repositorio remoto, para que eso sea posible hay que darle acceso al nuevo usuario al repositorio, para eso en settings(del repositorio)🡪collaborators y se le agrega el email del nuevo colaborador(solo si es publico) o el nombre de usuario, el nuevo colaborador debe aceptar la invitación.

**Flujo de trabajo:**

Por norma general como una buena práctica no se agregan archivos binarios a los repositorios, como los formatos png, la razón de esto es los archivos binarios son pesados, y siempre que se vayan a hacer cambios, se trae ese cambio, y mientras más binarios más pesado se vuelve el repositorio. Por ejemplo, si se le hace un cambio a una imagen, github no necesariamente actualiza el archivo en el repositorio, o se demorara en actualizar, porque los archivos binarios son raros, para forzar la actualización se con ctrl+shift+r o con ctrl+f5. La rama master solo se envía cuando se esta seguro que está listo para producción. Con git add imágenes/imagen.png se agrega un archivo de tipo imagen que se encuentra en la carpeta imágenes, y con git commit -am “logo” se agrega al repositorio, como cualquier otro commit, un git pull origin header para traerme la ultima versión, y un git puch origin header para enviar la imagen al repositorio remoto. (el proceso es básicamente el mismo de un commit normal)

Para un buen flujo de trabajo, cada programador no debería hacer commits a master hasta que se esté completamente seguro, es mejor realizar commits a las ramas dentro del repositorio. Por ejemplo, para empezar a trabajar una rama hay que traérsela con pull origin footer, y luego posicionarme en la rama con git checkout origin footer, cuando se vayan a enviar los cambios es buena práctica hacer un git status para saber que archivos se modificaron. As probable que si se trabajan con deferentes sistemas operativos se presenten problemas con las tildes, porque el estándar de tildes de Mac es diferente al de Windows.

Es trabajo del líder de proyecto y esa gente fusionar las ramas para eso se vuelve a la rama master con git checkout master, y luego git merge header, después se envía al repositorio remoto con git pull origin master y luego git push origin master, después se hará el merge de la rama footer de igual forma.

**Pull requests:**

En un entorno profesional se bloquea la rama master, para que nadie le haga merge a master porque si, tiene que haber una revisión. La rama master es la que se va a subir al servidor de producción donde está la versión final, pero a veces hay que hacer pruebas en un entorno similar al entorno de producción, y que sea tan similar como sea posible sin afectar a los usuarios, esos lugares son servidores de desarrollo o servidores de staging, que son ramas justo antes de master y es un servidor de pruebas.

Funciona creando una rama especial donde estará lo que se esté desarrollando en este caso footer y header (un feature), y después la versión final de esa rama se envía a staging (el de prueba, y siempre debe estar actualizado), en vez de hacer un merge va a un pull request, que permite que otros miembros del equipo vean los cambios para aprobarlos, al aprobarlos se autoejecuta el merge en staging(el de prueba). Antes de fusionar los cambios con la rama maestra también necesita un pull requests y se hace después de haber probado los cambios. Los pull requests son una característica de github, y también permite a personas que no son colaboradores trabajar en el proyecto, el equipo que se encarga de administrar el entorno de desarrollo se llama DevOps.

Imagen que contiene Gráfico

Descripción generada automáticamente

**Utilizando un pull Request en github:**

Si existen problemas se puede crear una nueva rama para solucionarlos que luego se puede fusionar, esto es una buena práctica. Hacer un git pull master, después crear una nueva rama con git Branch fix-typo, y moverme a la nueva rama con git checkout fix-typo.

A continuación, se corrigen los problemas, y se hace un push a origin con git push origin fix-typo. Una ves la rama este en github se crea un nuevo pull request, se le dice que compare master con fix-typo, con git status muestra que un no se han enviado las correcciones al repositorio, así que se le hace el commit, y después de nuevo un git push origin fix-typo

Interfaz de usuario gráfica, Diagrama

Descripción generada automáticamente

Github determinara automáticamente que existen diferencias entre master, y la rama fix-typo. Para eso se le da un nuevo pull request, y se comparan las dos ramas, también se puede hacer automáticamente. Al hacer un pull request permite agregar detalles, y toma el nombre del commit como el nombre del pull request. Esto es útil en el proceso de devops, un proceso mediante el cual los programadores pueden trabajar mejor.

El pulll request no ejecuta un merge, solo describe los cambios, y lo que hace es decirle a alguien que eso no se debió haber hecho, desde la perspectiva de la persona que esta revisando los pull request (en el curso se llama anita) le aparecerá una notificación, que, si por favor hace la revisión de los cambios porque quiere fusionar un commit hacia master, github automáticamente detectara si hay conflictos, y poreguntara si se hace un merge commit, un Squash and merge o un Rebase and merge. (las últimas dos son malas prácticas porque ….)

Un pull request puede ser una rama entera, y se puede aprobar, comentar o pedir cambios. Que el pull request esté aprobado no significa que el merge se halla ejecutado, sino que está listo, alguien tiene que hacer el merge y que se respete, es una buena practica hacer siempre un code review. Para hacer el merge hay que darle “merge pull request” y confirmamos el merge. A partir de aquí se pude borrar el Branch, y github da la opción.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

(Las entidades html es más o menos &aacute;=á)

Si regresamos al entorno local y le damos un pull a la rama fix-typo va a decir que no existe, pero si le damos pull a la rama master, si aparecerá toda la historia, porque en git realmente no se borra nada, queda todo registrado. En el lado de git no existen los pull request, solo existen en github, es como una pausa justo antes de fusionarlo.

**Creando un fork:**

Los proyectos open source se pueden clonar, pero no se puede agregar nada a ellos. El primer paso para colaborar con un proyecto es darle watch, una estrellita y un fork. Hacer un fork es tomar una copia del estado actual del proyecto y clonarlo como un proyecto mio, es una característica única de github.

Se le da clic a fork y copiará todo el proyecto, y para poder hacer cambios hay que traer cambios el proyecto con clone or download y se copia el enlace, en la consola de git y se coloca el comando git clone (enlace), a partir de aquí se puede hacer cualquier cambio y se subirá al repositorio local como en cualquier momento, y también será posible hacer push al repositorio remoto, pero no al oficial, sino al clonado.

Para hacer el pull request para que ambos proyectos queden unidos, se le da a new pull request, aquí pedirá escoger las dos ramas para comparar, o dos fork para comparar, posteriormente aparecerán los cambios y pedirá el titulo y la descripción. (un draft pull request es esperar a ver si me da la gana de enviarlo) desde el punto de vista del dueño del repositorio aparecerá quienes siguen y clonaron el repositorio, también aparecerá el pull request y si hubiera algún conflicto, después se confirma el pull request, y se confirma el merge con la rama master del contribuidor y del dueño.

Algo interesante de los fork es que no se actualizan automáticamente, si un proyecto sigue avanzando los fork no lo harán a la par del proyecto, hay que traer los cambios que le hagan al repositorio original. Para actualizar un fork , primero mostrará que el repositorio original esta adelantado al fork, y se puede hacer un pull request del original al fork, para poder fusionarlo y actualizarlo, pero también se puede hacer desde consola, para eso hay que hay que crear otra fuente para hacer pull, eso quiere decir hay que copiar el link del proyecto original, y en la consola de git se coloca el comando git remote add upstream (enlace original), lo que crea otra fuente nueva para traer datos a la rama master. Finalmente se da un git pull upstream master para traerme todos los cambios del repositorio original, con git commit -am “fusión”, y luego un git push origin master para actualizar el blog en github.

Básicamente se traen los cambios al repositorio local los cambio directamente del repositorio maestro, y luego se envían al fork en github. También se puede hacer al revés, desde github.

**Haciendo un deployment a un servidor:**

Normalmente la rama master es la que se envía a producción. La forma de enviar a producción es entrando a la carpeta que contiene los archivos del servidor, y copiar la url de https que nos da github y le damos git clone (url). Después con el enlace se puede ver el archivo .html.

Se pueden hacer commits directamente desde github, pero no actualizara el archivo dentro del servidor, para eso se da git pull origin master en el mismo lugar donde están los archivos del servidor.

Con travis-ci es como un github que conecta automáticamente las ramas de github con los servidores, Jenkins también permite hacer eso y este es de instalar. A esto se le llama integración continua y desarrollo continuo en el mundo de DevOps.

**Hazme un pull request:**

\*\*En este clase hay que ir el hyperblog de freddier, y hacerle un fork en donde dentro del id=”post” y debajo de suscribete y dale like, ahí se agrega el nombre.

**Ignorando archivos (.gitignore):**

No todos los archivos de un proyecto deberían ir al repositorio, como por ejemplo el archivo de contraseñas. En git existe algo llamado gitignore. Supongamos que el blog tiene una nueva imagen, y con git status va a detectar que esta el archivo binario de la imagen y la modificación del html, vamos a crear un archivo nuevo que se llame .gitignore, este archivo va a ser una lista de archivos que vamos a ignorar. Para ignorar los jpg se escribe en el archivo da \*.jpg y se guarda, en git status ya no va a aparecer las imagen pero si el .gitignore, así que se agrega con git commit -am ”se agrega al blog una imgen”, después se envía al repositorio remoto con git pull origin master y se fusiona, después un git push origin master.

La imagen no va a quedar en el repositorio porque es una buena práctica que no hallan archivos binarios en el repositorio, las imágenes se deberían subir con ftp o rzinc.

Ahora hay que subir la imagen en imgur.com que deja subir imágenes gratis, para referenciar archivos binarios de otro lado en el sistema.

**Hablemos de readme.md:**

Md significa mark down, es una especie de código que permite cambiar ligeramente como se ve un texto, y readme.md existe para contarle al mudo para que es el repositorio, y siempre está en la raíz del repositorio, tenerlo es una muy buena práctica. Cuando lo abres ves un pedazo html. (un md también es como funciona Wikipedia, por ejemplo)

Para editarlo se puede usar un editor online o simplemente visual studio code. Un buen editor online se llama pandao (https://pandao.github.io/editor.md/en.html), este editor muestra directamente como de va a ver el readme.md en github, y simplemente se copia lo que se haga allá y se pega en el readme.md abierto en visual studio code. Después se hace un git pull origin master, luego un git commit -am “readme modificado para explicar el proyecto” y finalmente git push origin master. Así ya queda enviado al repositorio remoto y se verá en github.

**Sitio web público con github pages:**

Github tiene github pages que es un servicio de husting gratis, es tener un repositorio donde el contenido se valla a github y se vea online: pages.github.com

Para hacerlo hay que crear un repositorio con el nombre de usuario, que sea público. Ahora en la maquina local, en home hay que decirle git clone (url del repositorio) despues con ls -al va a aparecer una nueva carpeta con el nombre del repositorio. La guía dice que se crea un archivo llamado index.html, con vim index.html se crea y luego un git add index.html, para hacer el commit con git commit -am”archivo index”, para enviarlo al nuevo repositorio es con git pull origin master, y git push origin master. El ultimo paso es configurarlo, en settings habrá una opción de github pages, ahí se le dice que agarre el master Branch.

Finalmente el url será (nombre de usuario).github.io, pero para que cargue en la raíz el nombre hay que cambiarlo a (nombre de usuario).github.io

Esto también funciona con el repositorio que se viene trabajando, pero para eso hay que poner el archivo exacto con /hyperblog.

**Git rebase:**

Un rebase es una forma de unir la historia de dos ramas diferentes, y hacer como que no paso nada.



Esto es una mala practica para enviarla a repositorios remotos, porque la historia de repositorios remotos debería mantenerse limpia, esto porque rebase reescribe la historia de la rama master, como un parche.

Vamos a agregar un cambio en historia.txt con un commit, y ahora vamos a crear una nueva rama con git Branch experimento, y nos pasamos a esa rama con git checkout experimento, vamos a agregar otra línea a historia.txt y le damos commit a la rama experimento, y agregamos otra línea para que queden dos commits en la rama experimento. Ahora queremos pegar esas ramas y hacer como que nuca paso la rama experimento, para eso hay que estar en la rama experimento, y ponemos git rebase master, en este punto la rama master no abra cambiado, pero la rama experimento va a tener toda la historia de master. Ahora agregamos un nuevo commit a la rama master con un cambio de nuevo a historia.txt. de nuevo vamos a darle git rebase master desde la rama experimento, aquí va a contarnos cuales fueron los cambios que se hicieron. Básicamente cambia la historia de donde arranco el Branch.

Finalmente, un git checkout master, y un git rebase experimento y quedara ahora como si experimento hubiera existido desde siempre y siempre hubiera tenido el mismo contenido de la rama master (siempre se le hace rebase de experimento a master, y luego de master a experimento).

Ahora se envía todo al repositorio remoto: git pull origin master, git push origin master. En github aparecerá como si los commits de la rama experimento fueran los mismos de la rama master. Con git Branch -D experimento va a decir que la rama se borró. (aquí no queda historia, y no se sabe quién hizo que)

**Git stash para guardar cambios en memoria y recuperarlos después:**

Gitk nos muestra toda la historia de lo que se a hecho hasta ahora. Si por decir algo realizo un cambio por accidente y no se como volver a la versión original del ultimo commit, git stash me permite regresar el archivo a la última versión guardada en esa rama, los cambios que se hicieron “por accidente” están guardados en un lugar temporal, y con git stash list se puede ver que tengo algo stasheado.

Si quito el stash en otra rama, me saca el trabajo que se hizo en al footer por ejemplo, es mejor hacer un git checkout master, y un git stash pop se abre el stash que se tenía almacenado con los cambios hechos por “error”.

Otra cosa que se puede hacer es guardar los cambios y ponerlos en una rama, para eso realizamos cambios en el archivo html, y le damos git stash, y para poner eso en una rama ponemos git stash Branch (nombre de la rama). En git status me mostrara que el archivo esta modificado, aquí hacemos un commit para agregarlo a la nueva rama. Si volvemos a la rama master estará como se dejo en el último commit, y no aparecerá nada en stash.

¿Y qué pasa si realmente no quiero guardar el stash? Pues si no hubo commit se le da git stash list para restaurar la versión antes y con git stash drop va a eliminar el stash.

El stash es cuando se quieren hacer cambios pequeños que no ameriten una rama o cuando se lleva mucho trabajo adelantado y no se a hecho commit, y necesitas datos de otra rama. Es muy típico y un comando muy popular.

**Git clean: limpiar tu proyecto de archivos no deseados:**

A veces hay archivo que no se deberían agregar al repositorio, o que por error se terminan crenado y no se quieren agregar al repositorio, pues git sabe cuál es la estructura de trabajo, y si detecta archivos nuevos que no están en el repositorio y no se quieren agregar puedes poner git clean, aquí va a preguntar si es en serio, así que se le da git clean—dry--run para que muestre los archivos que va a borrar. Con git clean -f borra definitivamente los archivos, pero van a quedar las carpetas, porque a git le vale madres las carpetas, así que se deben borrar a mano. Por otra parte las imágenes tampoco se van a borrar pero porque en .gitignore están seleccionados todos los archivos jpg. O sea que git solo borra las cosas que puede indexar.

**Git Cherry-pick: Traer commits viejos al head de un branch:**

Digamos que necesitas los avances de una rama, pero no toda la rama sino un commit en concreto, para eso existe Cherry-pick. Vamos a modificar el readme para agregarle cosas y se va a guardar para ponerlo en otra rama. Vamos a hacer un git stash, y luego un git stash Branch readme-mejorado para crear una nueva rama, y para agregar el commit le damos un git commit -am ”readme mejorado” (git log --oneline se ven los datos de los commits de forma más reducida). Ahora vamos a hacer unos nuevo cambio y le hacemos de nuevo dos commit en la rama readme-mejorado.

Ahora nos devolvemos a master, y solo queremos los cambios del primer commit, para eso le damos git Cherry-pick (el hush del commit que se quiere traer). Aquí no se tiene que hacer un git add ni un git commit, y en el historial aparecerá automáticamente el commit, como si hubiera ocurrido en master. Finalmente le damos git merge readme-mejorado, aquí va a aparecer un conflicto porque ya me traje una línea antes, y simplemente se soluciona como cualquier otro conflicto, se guarda en code, y se le hace commit a master.

Hay que recordar que la rama readme-mejorado es una rama local, nunca se envió al repositorio remoto. El cerry-pick es una mala práctica, porque significa que estas reconstruyendo la historia, también funciona en cualquier rama.

**Reconstruir commits en git con amend:**

A veces se envían commits por error, así que supongamos que se hace un commits con un cambio al archivo html, pero se olvidó otro cambio, en este caso se realiza el cambio que faltaba, se guarda y se le da git add, y con git commit --amend se puede agregar el cambio al commit que ya se realizó, aquí preguntara si se quiere cambiar al texto del commit.

**Git Reset y Reflog para emergencias:**

¿Y si todo se fue al carajo? Como si hubieras borrado por accidente un archivo y se le hubiera hecho un commit, o se hizo un merge erróneo o algo así, en estos casos existe un comando que no olvida nada, primero se ejecuta git reflog, aquí va a mostrar todo y simplemente se busca ultimo head en donde todo era correcto, aquí se copia ese head, y se le da git reset (y la cabeza que está bien), la diferencia entre soft y hard es que en soft va a mantener lo que este en staging y hard si resetea todo. Cuando se ejecuta el comando no va a aparecer el archivo eliminado, aparecerán los cambios en staging.

Para poder hacer un reset hard, se le da git reflog de nuevo y copiamos el numero amarillo antes de que tono se fuera a la m\*\*\*da, después se le da git reset –hard (numerito del head), y ahora todo volvió a la normalidad, en git log va a desaparecer los cambios que no se querían. Por eso reset es muy peligroso, porque cambia la historia del repositorio, y normalmente es una mala práctica, y con git reflog si guarda la verdad y desde ahí se pueden volver a traer lo errores.

**Buscando archivos y commits en git con Grep y log:**

A medida que el proyecto se hace grande, se van a necesitar buscar ciertas cosas, con git grep color va a buscar la palabra color en los archivos y mostrará en que archivo esta, con git grep -n color mostrará los mismo que en el caso anterior, pero dirá exactamente la línea del archivo en donde esta. Este comando se puede usar con cualquier palabra. También se puede contar la cantidad de veces que esta una palabra, para eso es git grep -c la, y va a contar cuantas veces esta la palabra la y en que archivo. Por ejemplo, para buscar una etiqueta, se hace git grep -c “<p>”.

Para buscar palabras en la historia de los commits se hace git log -S “cabecera” y buscara todas las veces que se uso cabecera en los commits.

Grep para los archivos log para los commit.

**Comandos y recursos colaborativos en git y github:**

Harry Roberts creo unos comandos para manejar el equipo de una manera mas colaborativa. El primero es para ver cuántos commits hizo cada miembro del equipo, este es git shorlog, con git shortlog -sn solo va a mostrar las personas que han hecho ciertos commits, y con git shortlog -sn --all va a mostrar todos los commits incluyendo los que fueron borrados, y por último git shortlog -sn --all --no-merges va a mostrar los commits sin los merge. Ahora vamos a crear un nuevo comando con git config –global alias.stats “shortlog -sn --all --no-merges ”, ahora con git stats va a mostrar las estadísticas de los commits.

Con git blame y el archivo podemos ver quien modifico que, en determinado archivo, el git blame -c y el archivo va a mostrar mejor la información. Se puede especificar las líneas más o menos así git blame (archivo) -L35,53 y va a mostrar solo esas líneas y quien las cambio.

Con el comando que se quiera agregando --help va a abrir el navegador con el manual de cómo funciona el comando.

Finalmente, para ver ramas remotas con git Branch -r se ven las ramas remotas que están en github. En git Branch -a va a mostrar en blanco las locales y en rojo las remotas. En github va a existir un panel llamado insights que va a mostar la información de los contribuyentes y esas cosas, como forks y todo eso, se puede ver por semanas o meses, esto funciona para cualquier repositorio.

**¿Y ahora?**

Existe un ciclo de vida del desarrollo de software, ver que es Travis o Jenkins. Gitlab? DevOps?

Diagrama

Descripción generada automáticamente